Contents

1 Summary

This report details the two-semester UROPS project entitled "Parking Lot Classification"; namely, it describes the suite of tools developed and results achieved in the effort to use convolutional neural networks (CNNs) to ascertain the state of a parking lot given its picture. The report will briefly recount the first semester's results and intentions before delving into their development in the second. It will conclude with an evaluation of the soundness of using CNNs to report parking lot occupancy in the practical context.

2 Recount of ST2288

The motivation behind investigating the use of CNNs for this task was to determine whether it was possible to reliably and inexpensively increase the resolution of information available to drivers looking for parking space; in Singapore, most public parking lots provide users with information on the *number* of spots left, but do not tell them *where* empty spots are. When parking is scarce, traversal to find free spots can be a great inconvenience. Thus, an Internet-of-Things idea based on the usage of already-installed CCTV cameras was proposed to be tested:

how can one use the *image* of a parking lot to determine its spot-wise occupancy, and then deliver this information to users, in real-time?

CNNs were selected to be the classification model behind this idea's implementation since they are currently state-of-the-art for image classification tasks.

The following is a possible workflow of a system based on this idea:

1. User requests for spot-wise occupancy status of a given parking lot through a mobile application created to handle such requests in real-time.
2. A picture of the lot is taken by a pre-existing CCTV or dedicated camera, and transmitted to the associated cloud-based CNN.
3. Each spot in the parking lot is classified by this CNN as either empty or occupied.
4. The picture is then deleted, and the is user sent a spatially-accurate abstraction of spot-wise occupancy status, possibly in the manner pictured below.

Figure 1: This visual could be delivered to users through the mobile application.

The focus of work in ST2288 was solely on the third step of the above workflow: to explore and determine the effectiveness of CNNs for this classification task. To this end, CNNs were found to work very well. A publicly-available parking lot image dataset, *PKLot*[1], was used to train, validate, and test CNNs created. It

[1]This dataset can be found at: web.inf.ufpr.br/vri/databases/parking-lot-database/

comprises of 695,899 images of parking **spots** taken from two parking lots over the course of about 30 days, in which there was great variation of weather and illumination [1]. The following are three examples from this dataset:

Figure 2: An occupied spot from the first parking lot, in sunshine.

Figure 3: An empty spot from one angle of the second parking lot, in overcast conditions.

Figure 4: An occupied spot from the second angle of the second parking lot, in the rain.

The testing accuracies of the CNNs created for each lot in this dataset were greater than 99.8%, which translates to the misclassification of about 260 spots in requesting for the prediction of the state of about 175,000. The following was the CNNs' common architecture:

(Learning rate: 0.001)

Input layer: reads in 32-by-32 color images of parking spots.

Convolutional layer 1: applies 32 5-by-5 filters, and then the rectified exponential linear unit (ReLU) activation function.

Pooling layer 1: performs max pooling with a 2-by-2 filter.

Convolutional layer 2: applies 64 3-by-3 filters, and then the ReLU activation function.

Pooling layer 2: performs max pooling with a 2-by-2 filter.

Fully-connected layer 1: comprises of 1,024 neurons.

Output layer: comprises of 2 neurons, representing the output vector.

$(0, 1) \rightarrow$ occupied spot

$(1, 0) \rightarrow$ empty spot

The CNNs were implemented using TensorFlow, an efficient and well-supported deep-learning library written for the Python programming language. Their training and evaluation was done online on FloydHub. FloydHub is a Platform-as-a-Service that quite inexpensively offers a pre-configured programming environment on powerful hardware for machine learning purposes. Users run "jobs" through a command-line client, with data and algorithms stored on FloydHub. Training metrics and logs can be automatically generated:

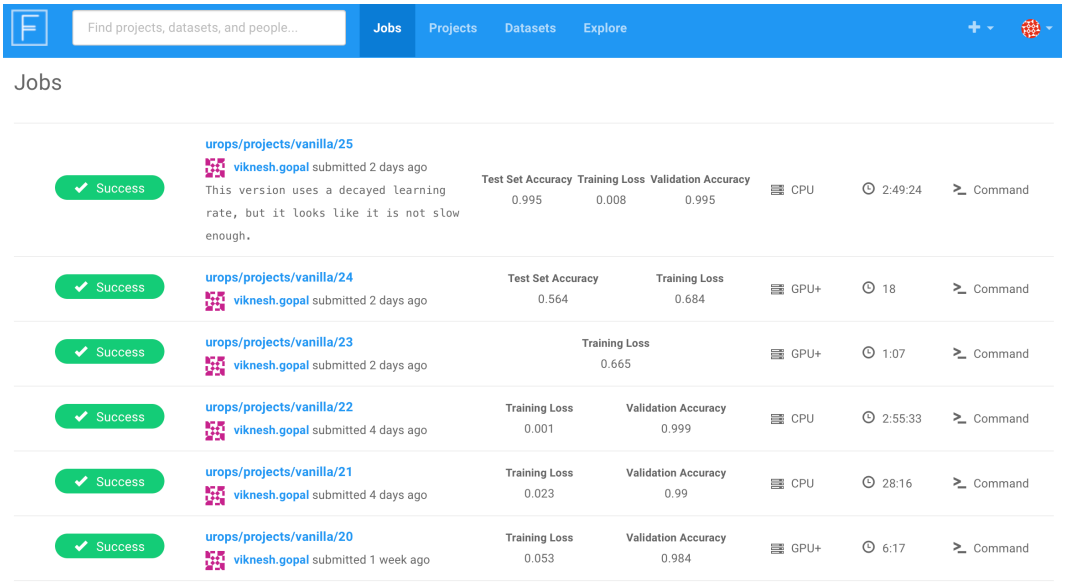

Figure 5: Log of jobs run on FloydHub.

With the above results and tools, ST2288 was concluded. This left trialling of the practical application of this idea in the local context to ST3288. The rest of this report details the tools created in response to the challenges involved in the creation and management of original data and the readily-extensible CNN implementation developed. The files referred to in the rest of this report are as they appear on github.com/nurmister/urops.

3 Creation of a Singaporean *PKLot*: *NUSLot*

3.1 Collection schedule and setup

NUSLot is the culmination of the data collection, processing, and labeling efforts of this project. It comprises of 50,000 labeled examples of spots of the parking

lot belonging to block S17 at the Faculty of Science. The following are three examples from this dataset:

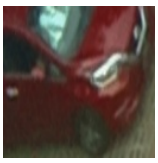

Figure 6: An occupied spot.

Figure 7: An empty spot.

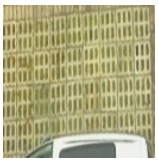

Figure 8: Another empty spot. Notice the occlusion caused by the roof of the vehicle occupying the adjacent spot.

More specifically, this dataset consists of 50,000 128-by-128 pixel BGR JPEG images of parking spots, taken between the fourteenth of June and the 24th of July. These images were taken from the following vantage:

Figure 9: A Raspberry Pi-based camera was mounted on this window,

Figure 10: located at the encircled position,

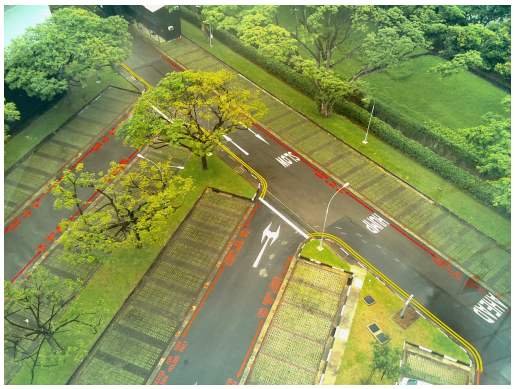

Figure 11: facing the following parking lot.

Pictures were taken every five minutes by a Raspberry Pi-based camera, as pictured below:

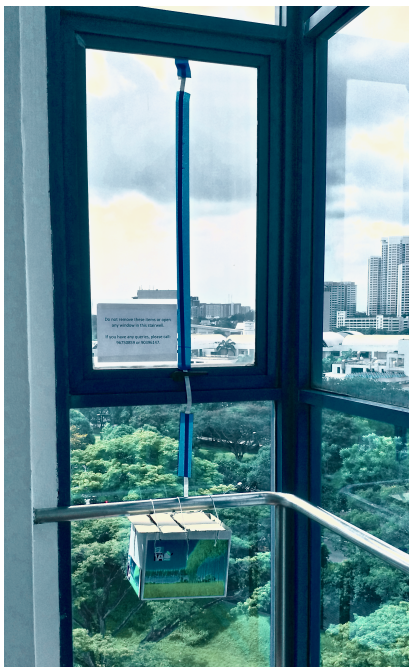

Figure 12: The Raspberry Pi, its memory, and its power supply were enclosed in the cooled box.

Figure 13: The camera module was mounted firmly on the window at a suitable angle.

The following were the apparatus used:

1. Raspberry Pi Model B+, running Raspbian 4.14.

   Power supply: 20,000 mAh USB battery pack, enclosed in a LiPo battery blast protector.

   Memory: External 64 GB USB drive.

2. Raspberry Pi Camera Module V1.

   Connected to the Raspberry Pi using a two-meter flex cable.

3. Portable fan, modified to run off of a 10,0000 USB battery pack (also enclosed in a LiPo battery blast protector).

The battery packs lasted about a day given the high temperatures at the vantage. The Raspberry Pi and battery packs were thus removed each evening and replaced the subsequent morning.

3.2 Cropping of spots from complete images

The pictures the Raspberry Pi took were of the complete parking lot: thus, in the evenings, these pictures were cropped into images of individual spots. The cropping process was fully automated using the scripts of `label_examples/crop_spots/` following a one-time setup of `label_examples/crop_spots/crop_instructions.csv`. These instructions were used by `label_examples/crop_spots/crop_all_spots.py`, which is itself called by `label_examples/crop_spots/daily_cropper.sh`. The latter script is what the end-user has to call to automatically crop all images placed in `label_examples/pictures_dump`.

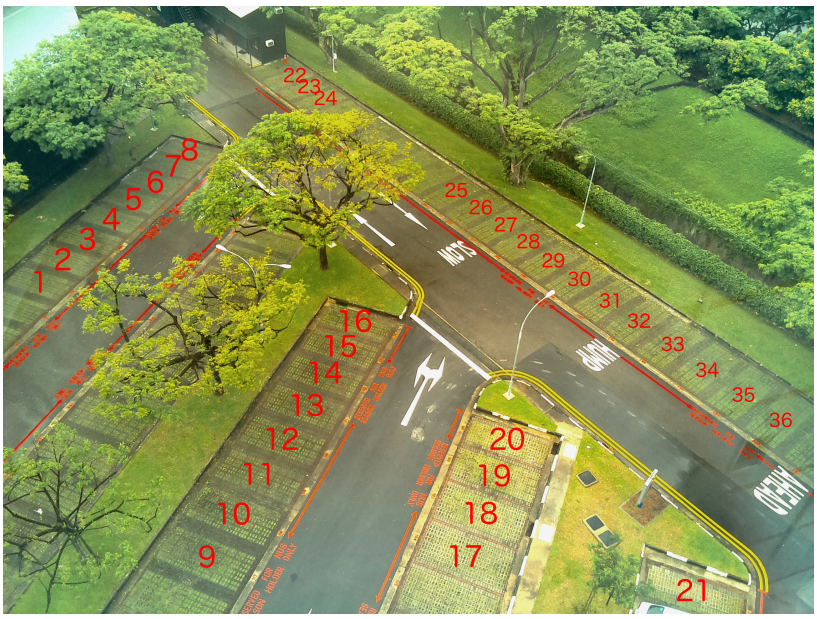

Figure 14: Each spot in the parking lot was assigned an ID for cropping and file-naming purposes.

For each spot, these cropping instructions were created using `label_examples/crop_spots/get_spot_coords_and_angles.py`. This script accepts as flags the path to an image and an angle to rotate the image counter-clockwise by. It then displays the rotated image and allows users to click-and-drag green bounding boxes over it to ascertain the coordinates of the region to crop the image to obtain a picture of the desired spot. It prints the coordinates of drawn bounding boxes in the terminal from which the script is called. The following is an example of its use:

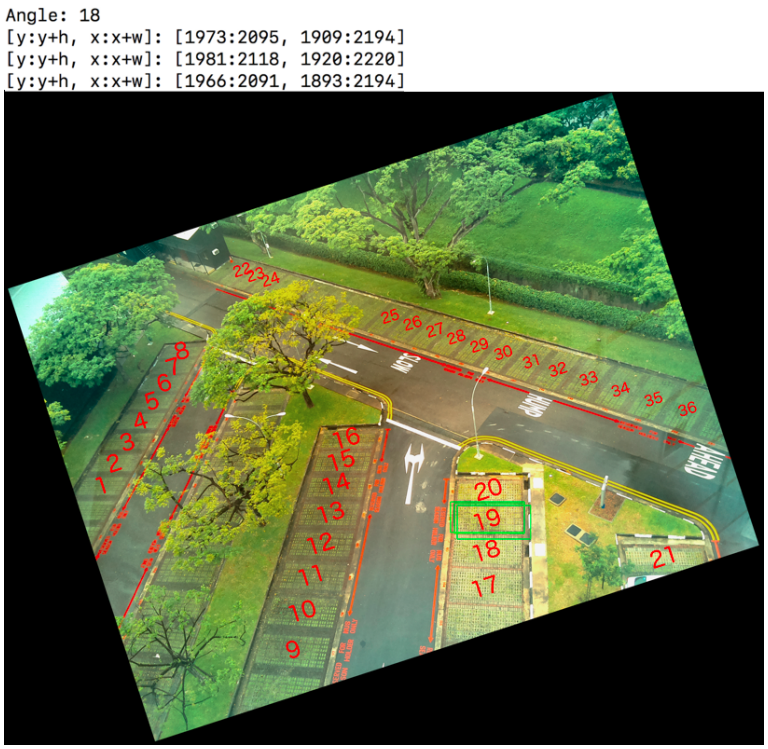

Figure 15: An attempt to obtain the right angle and coordinates to crop images for spot 19.

The following was thus the cropping workflow:

1. Images to be cropped are placed in `label_examples/pictures_dump`, and `label_examples/crop_spots/daily_cropper.sh` is called.
2. The script calls `label_examples/crop_spots/crop_all_spots.py`, which pipes shell commands to obtain the image of each spot from each image to be cropped into `label_examples/crop_spots/todo.sh`. These shell commands are calls to `label_examples/crop_spots/crop.py`, which crops and saves the image of a specified spot from a single image, given the spot's cropping instructions.

3. `label_examples/crop_spots/todo.sh` is deleted.

Thus, apart from manual determination of each spot's cropping instructions, the user is insulated from the otherwise tedious cropping process.

```
python crop.py --image 2018-07-24-1020.jpg --angle 22 --x_one 1327 --x_two 1543 --y_one 2000 --y_two 2076 --label 13
python crop.py --image 2018-07-24-1020.jpg --angle 22 --x_one 1346 --x_two 1559 --y_one 1889 --y_two 1962 --label 14
python crop.py --image 2018-07-24-1020.jpg --angle 22 --x_one 1372 --x_two 1574 --y_one 1787 --y_two 1851 --label 15
python crop.py --image 2018-07-24-1020.jpg --angle 22 --x_one 1388 --x_two 1581 --y_one 1707 --y_two 1756 --label 16
python crop.py --image 2018-07-24-1020.jpg --angle 22 --x_one 1939 --x_two 2239 --y_one 2220 --y_two 2353 --label 17
python crop.py --image 2018-07-24-1020.jpg --angle 22 --x_one 1947 --x_two 2228 --y_one 2083 --y_two 2201 --label 18
python crop.py --image 2018-07-24-1020.jpg --angle 21 --x_one 1943 --x_two 2209 --y_one 1969 --y_two 2076 --label 19
python crop.py --image 2018-07-24-1020.jpg --angle 21 --x_one 1943 --x_two 2205 --y_one 1867 --y_two 1947 --label 20
python crop.py --image 2018-07-24-1020.jpg --angle 20 --x_one 2669 --x_two 2867 --y_one 2125 --y_two 2201 --label 21
python crop.py --image 2018-07-24-1020.jpg --angle -20 --x_one 1464 --x_two 1570 --y_one 775 --y_two 821 --label 22
python crop.py --image 2018-07-24-1020.jpg --angle -20 --x_one 1490 --x_two 1612 --y_one 829 --y_two 867 --label 23
python crop.py --image 2018-07-24-1020.jpg --angle -20 --x_one 1524 --x_two 1646 --y_one 874 --y_two 916 --label 24
python crop.py --image 2018-07-24-1020.jpg --angle -25 --x_one 1844 --x_two 1973 --y_one 1319 --y_two 1372 --label 25
python crop.py --image 2018-07-24-1020.jpg --angle -25 --x_one 1882 --x_two 2026 --y_one 1391 --y_two 1452 --label 26
python crop.py --image 2018-07-24-1020.jpg --angle -25 --x_one 1939 --x_two 2068 --y_one 1471 --y_two 1532 --label 27
python crop.py --image 2018-07-24-1020.jpg --angle -25 --x_one 1973 --x_two 2129 --y_one 1555 --y_two 1616 --label 28
python crop.py --image 2018-07-24-1020.jpg --angle -25 --x_one 2023 --x_two 2182 --y_one 1642 --y_two 1699 --label 29
python crop.py --image 2018-07-24-1020.jpg --angle -25 --x_one 2076 --x_two 2239 --y_one 1734 --y_two 1794 --label 30
python crop.py --image 2018-07-24-1020.jpg --angle -27 --x_one 2125 --x_two 2285 --y_one 1851 --y_two 1920 --label 31
python crop.py --image 2018-07-24-1020.jpg --angle -29 --x_one 2171 --x_two 2334 --y_one 1973 --y_two 2049 --label 32
python crop.py --image 2018-07-24-1020.jpg --angle -31 --x_one 2209 --x_two 2372 --y_one 2102 --y_two 2190 --label 33
python crop.py --image 2018-07-24-1020.jpg --angle -33 --x_one 2236 --x_two 2403 --y_one 2251 --y_two 2353 --label 34
python crop.py --image 2018-07-24-1020.jpg --angle -32 --x_one 2304 --x_two 2471 --y_one 2369 --y_two 2464 --label 35
python crop.py --image 2018-07-24-1020.jpg --angle -36 --x_one 2289 --x_two 2468 --y_one 2559 --y_two 2654 --label 36
python crop.py --image 2018-07-24-1035.jpg --angle 17 --x_one 410 --x_two 543 --y_one 1851 --y_two 1931 --label 1
python crop.py --image 2018-07-24-1035.jpg --angle 18 --x_one 460 --x_two 608 --y_one 1772 --y_two 1844 --label 2
python crop.py --image 2018-07-24-1035.jpg --angle 18 --x_one 517 --x_two 657 --y_one 1692 --y_two 1756 --label 3
python crop.py --image 2018-07-24-1035.jpg --angle 18 --x_one 566 --x_two 703 --y_one 1612 --y_two 1673 --label 4
python crop.py --image 2018-07-24-1035.jpg --angle 18 --x_one 616 --x_two 749 --y_one 1532 --y_two 1593 --label 5
python crop.py --image 2018-07-24-1035.jpg --angle 18 --x_one 654 --x_two 794 --y_one 1460 --y_two 1517 --label 6
python crop.py --image 2018-07-24-1035.jpg --angle 17 --x_one 695 --x_two 836 --y_one 1384 --y_two 1433 --label 7
python crop.py --image 2018-07-24-1035.jpg --angle 17 --x_one 741 --x_two 870 --y_one 1323 --y_two 1365 --label 8
python crop.py --image 2018-07-24-1035.jpg --angle 21 --x_one 1175 --x_two 1433 --y_one 2536 --y_two 2654 --label 9
python crop.py --image 2018-07-24-1035.jpg --angle 22 --x_one 1232 --x_two 1475 --y_one 2384 --y_two 2498 --label 10
python crop.py --image 2018-07-24-1035.jpg --angle 22 --x_one 1262 --x_two 1505 --y_one 2247 --y_two 2350 --label 11
python crop.py --image 2018-07-24-1035.jpg --angle 22 --x_one 1300 --x_two 1521 --y_one 2114 --y_two 2201 --label 12
python crop.py --image 2018-07-24-1035.jpg --angle 22 --x_one 1327 --x_two 1543 --y_one 2000 --y_two 2076 --label 13
python crop.py --image 2018-07-24-1035.jpg --angle 22 --x_one 1346 --x_two 1559 --y_one 1889 --y_two 1962 --label 14
```

Figure 16: `daily_cropper.sh` generates shell commands to crop all desired images.

3.3 Labeling of cropped images

After cropping, the resulting images are transferred to `label_examples/label_spots/pictures_to_label/` for labeling, which is done using `label_examples/label_spots/spot_labeler.R`. This script displays each cropped image in sequence, prompting the user to type a label for each one. In the case of this problem, labels were either "0" or "1", depending on whether the spot was empty or occupied (respectively). Once each image has been labeled, labels are written to a CSV file entitled by the date the images were taken. The following is a visual of the labeling process:

Figure 17: The image of each spot is displayed in sequence.

3.4 Serialization of data

After the data has been collected, processed, and labeled, it is converted into NumPy arrays and serialized using `save_dataset/serialize_dataset.ipynb`. The following is the process:

1. All available cropped images and labels are used to create the feature and label NumPy arrays.
2. The set of examples is split up randomly into training and test sets, with the test set comprising of 10% of the total number of examples at hand.
3. Mean-subtraction and normalization of the test set is conducted using the mean and standard deviation of the training set. The test set is then saved as a HDF5 file in `data/hdf5/` as `test_set.hdf5`.
4. The training set is then further split into training and validation sets three times in a **shuffled, stratified** manner to allow for three-fold cross valida-

tion. For each split, the training and validation sets are mean-subtracted and normalized using the mean and standard deviation of the training set. This is done to help the CNN's gradient descent converge faster [3]. Each split is then saved to `data/hdf5/` as `train_validation_set_{split_number}.hdf5`, where `split_number` is either `1`, `2`, or `3`.

Three splits were made instead of the more common five or ten because of memory constraints: each split of the training set is 17.7 GB. Given that these scripts may be used to process other data, `save_data/serialize_data.ipynb` has been designed to be able to create a variable number of splits and handle a variable number of label classes and image types. (In fact, all aforementioned scripts can also be used for general data-processing given minor adjustment for the specific data at hand.) Also to note is that the third-party `h5py` module was used to serialize the data into HDF5 files instead of the more commonly-used inbuilt `pickle` module since the former is far more RAM-efficient in the serialization of arrays [2].

3.5 Culmination

After the completion of the above, the dataset is ready to use for machine learning purposes. It has been uploaded to Google Drive as `NUSLot`[2], and comprises of three directories:

1. `raw_data` comprises of complete parking lot images, cropped images of individual spots, and all label CSV files.

[2]It can be downloaded from goo.gl/fV2NXS.

2. `full_dataset` comprises of serialized NumPy arrays representing training, validation, and testing sets created from all 50,000 examples.

3. `toy_dataset` is a 10% simple random sample of the data of `full_dataset`.

4 Implementation of the CNN

The CNN was implemented using TensorFlow version 1.9's low-level API to allow for greater extensibility and functionality, which will be described later in this section. Firstly, however, a description of the particular CNN used for this classification problem.

4.1 Performance on *NUSLot*

Architecture:

(Learning rate: 0.0001)

Input layer: reads in 128-by-128 color images of parking spots.

Convolutional layer 1: applies 64 5-by-5 filters, and then the scaled exponential linear unit (SeLU) activation function [3].

Dropout is applied with a keep probability of 0.95.

Pooling layer 1: performs max pooling with a 2-by-2 filter.

Convolutional layer 2: applies 128 3-by-3 filters, and then the SeLU activation function.

Dropout is applied with a keep probability of 0.95.

Pooling layer 2: performs max pooling with a 2-by-2 filter.

[3] Using SeLU was found to lead to the fastest-learning CNN, reiterating past results: [4].

Convolutional layer 3: applies 256 3-by-3 filters, and then the SeLU activation function.

Dropout is applied with a keep probability of 0.95.

Pooling layer 3: performs max pooling with a 2-by-2 filter.

Convolutional layer 4: applies 512 3-by-3 filters, and then the SeLU activation function.

Dropout is applied with a keep probability of 0.95.

Pooling layer 4: performs max pooling with a 2-by-2 filter.

Convolutional layer 5: applies 1024 3-by-3 filters, and then the SeLU activation function.

Dropout is applied with a keep probability of 0.95.

Pooling layer 5: performs max pooling with a 2-by-2 filter.

Convolutional layer 6: applies 2048 1-by-1 filters, and then the SeLU activation function.

Dropout is applied with a keep probability of 0.95.

Pooling layer 6: performs max pooling with a 2-by-2 filter.

Fully-connected layer 1: comprises of 2048 neurons.

Dropout is applied with a keep probability of 0.90.

Output layer: comprises of 2 neurons, representing the output vector.

$(0, 1) \rightarrow$ occupied spot

$(1, 0) \rightarrow$ empty spot

This network, trained over 20 epochs, has a testing accuracy of 99.9%, a specificity of about 1, and a sensitivity of 0.998. This translates to the following confusion matrix:

$$\begin{pmatrix} 43348 & 1 \\ 3 & 1648 \end{pmatrix}$$

where the element at (0, 1), for example, represents the number of false positives. Given that there is only one such prediction, it is heartening to note that the network learned to correctly classify the large number of examples with occlusion in this dataset:

Figure 18: One of the many instances of occlusion that the network correctly classified as "empty".

These findings reiterate the conclusion of ST2288: CNNs are effective for this binary classification task, likely due to the low intra-class and high inter-class variation between examples.

4.2 Functionalities of this implementation

What is perhaps more significant than this performance is the TensorFlow implementation of this CNN and its extensibility; most readily-available implementations of CNNs tend to not be so. The following are some of its more novel features:

1. Automatic data management: the dataset to be used for training and evaluation – `toy_dataset`, `full_dataset`, or some other – is controlled via a flag in the function call. Moreover, the download and use of new datasets in the aforementioned HDF5 train/evaluation format can also be activated and managed by the implementation through specification of

another flag[4].

2. Dynamic scaling of network depth: most readily-available TensorFlow CNN implementations have a hard-coded architecture, and require a deep-dive into the code for changing network depth[5]. This implementation has been programmed in such a manner that only an integer representing the number of hidden layers and lists corresponding to filter details need to be supplied for the creation of the CNN. This feature makes model alteration far more convenient and less error-prone.

3. Ability to write evaluation mistakes to disk: statistics like loss and specificity only go so far in helping users gauge the performance of their network. It can often help to see the actual misclassified examples: they may reveal systemic weaknesses in the model's ability. This implementation thus has the functionality to write misclassified examples to a directory in the format `{spot_ID}_t-{true_label}_p-{predicted_label}`. For example, the following example was written to disk as `24_t-1_p-0.jpg`.

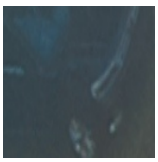

Figure 19: One of the three examples falsely classified as "empty".

[4]Guiding examples are provided in this project's GitHub repository to allow for quick setup of this data download option for any custom dataset.

[5]Consider the "canonical" github.com/aymericdamien/TensorFlow-Examples and tensorflow.org/versions/r1.0/get_started/mnist/pros

4. Seamless model saving and restoration options: this implementation can also save models created after a session of training, and then, given their file path, restore them in a new session. This allows for training to be split across sessions. While this functionality was not essential for the training of CNNs on this dataset, such training schemes are often used when training on very large datasets with millions of examples.

5. More options for dropout application: dropout is most commonly only applied after the first fully-connected layer, but there can also be benefits to applying dropout after convolution or after pooling [6]. Therefore, this implementation gives users the option to apply dropout in either way with the specification of one flag.

Apart from these more novel features, this implementation also has the following functionalities:

1. Encapsulated hyperparameter tuning: the end-user needs only to specify the numerical values for the model's hyperparameters; application of these choices to the construction of the computational graph is automated. This allows for more convenient model tuning.

2. Basic integration with TensorBoard, a visualization suite for TensorFlow: TensorBoard plots the computational graph of the network for visualization and debugging purposes, and also displays the accuracy and loss of CNNs being trained in real-time. This allows for easy comparison of different models.

Figure 20: Accuracy (top) and loss (bottom) comparison of two runs: blue with ReLU activation, and red with exponential linear unit (eLU).

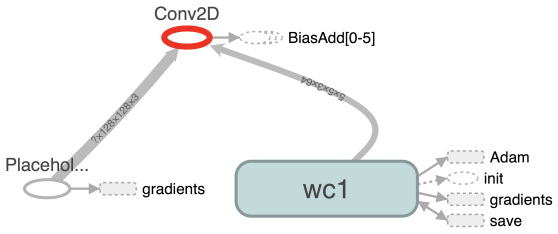

Figure 21: TensorBoard graph of the first convolutional layer.

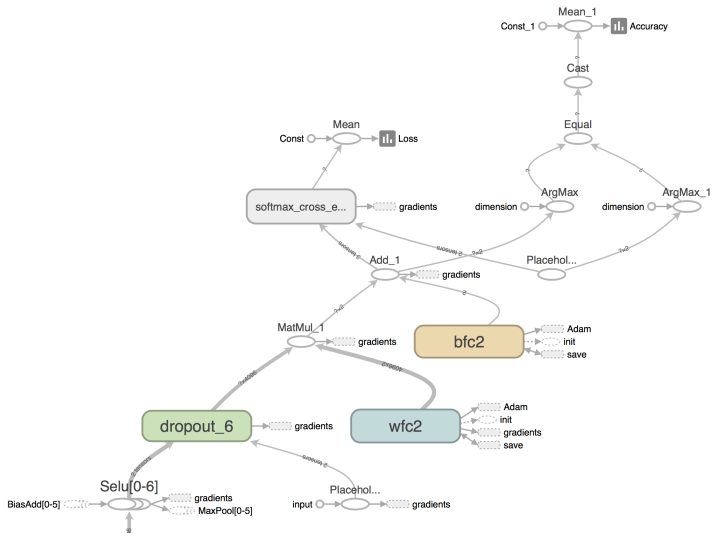

Figure 22: The tail-end of the complete computational graph.

A possible extension to current TensorBoard integration would be to enable plotting of layer activations across training epochs in the manner of [5]. This would aid in the comparison of activation functions.

3. Choice of several activation functions: users can choose between the ReLU, SeLU, eLU, leaky ReLU, cap-6 ReLU, softsign, and softplus activation functions with the use of just a flag.

4. Enhanced evaluation metrics: apart from displaying training loss and accuracy, this implementation also prints evaluation loss and accuracy, along with a generalized confusion matrix. If the number of classes in the dataset is two, the sensitivity and specificity of the model is also printed.

```
Training and validation on split 3.
------------------------------------------------------------
Epoch 1 | training loss:  0.53395, training accuracy:  0.92188.
Epoch 2 | training loss:  0.16924, training accuracy:  0.96094.
Epoch 3 | training loss:  0.10130, training accuracy:  0.96094.
Epoch 4 | training loss:  0.08622, training accuracy:  0.98047.
Epoch 5 | training loss:  0.06723, training accuracy:  0.98828.
Epoch 6 | training loss:  0.04625, training accuracy:  0.98438.
Epoch 7 | training loss:  0.03781, training accuracy:  0.98438.
Epoch 8 | training loss:  0.04027, training accuracy:  0.98828.
Epoch 9 | training loss:  0.03722, training accuracy:  0.98828.
Epoch 10 | training loss:  0.02591, training accuracy:  0.99219.
------------------------------------------------------------
Validation loss:  0.24588, validation accuracy:  0.94263.
Confusion matrix (true vs predicted):
[[991.   0.]
 [ 86. 422.]]
Specificity:  1.00000
Sensitivity:  0.83071
------------------------------------------------------------

------------------------------------------------------------
Validation accuracy (K = 3):
Mean:  0.93445
Median:  0.93600
Standard deviation:  0.00739
------------------------------------------------------------
```

Figure 23: Truncated display of performance metrics at the end of training and validation, with final validation accuracy calculated across all three splits.

Given this implementation's exceptional functionality and versatility, and its holistic compatibility with the aforementioned "farm to table" data processing

tools, the most significant contribution of this project may be less the evaluation of the soundness of using CNNs to ascertain parking lot occupancy and more the creation of open-source tools that can very easily be modified to solve other machine learning problems.

5 Conclusion

While CNNs are effective for ascertaining occupancy, the data collection process revealed that this idea is impractical due to the difficulty of obtaining suitable images for classification. Namely, a high vantage was needed to shoot pictures of this small parking lot to minimize occlusion; such a vantage likely does not exist or is not accessible for most lots. Moreover, even with such a high vantage, certain spots could not be used in classification due to tree cover. The hardware itself had high power consumption due to its operating environment, rendering it unsuitable for "hands-free" operation. Lastly, only daytime pictures could be taken and classified – most affordable cameras, including that of the Raspberry Pi, cannot take sufficiently sharp images during nighttime for the purposes of classification. In-ground sensors may be more favorable for this task.

Nonetheless, this project served as an excellent capability-building exercise: it made explicit the challenges to overcome and workflow required to efficiently create a dataset and prepare it for modeling upon. I therefore hope that the culmination of this project informs those carrying out their own with practical lessons, and provides them with a suite of tools easily modified to their own purposes.

References

1. Almeida, P. R., Oliveira, L. S., Britto, A. S., Silva, E. J., & Koerich, A. L. (2015). PKLot–A robust dataset for parking lot classification. *Expert Systems with Applications*, 42(11), 4937-4949. doi:10.1016/j.eswa.2015.02.009

2. Finch, C. (2010, January 10). Storing large Numpy arrays on disk: Python Pickle vs. HDF5 [Web log post]. Retrieved August 6, 2018, from https://shocksolution.com/2010/01/10/storing-large-numpy-arrays-on-disk-python-pickle-vs-hdf5adsf/

3. Ioffe, S., & Christian, S. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. doi:arXiv:1502.03167

4. Pedamonti, D. (2018). Comparison of non-linear activation functions for deep neural networks on MNIST classification task. Retrieved August 6, 2018, from https://arxiv.org/abs/1804.02763v1 arXiv:1804.02763v1

5. S. (n.d.). Activation-Visualization-Histogram. Retrieved August 7, 2018, from https://github.com/shaohua0116/Activation-Visualization-Histogram. Repository on GitHub.

6. Srivastava, Nitish & Hinton, Geoffrey & Krizhevsky, Alex & Sutskever, Ilya & Salakhutdinov, Ruslan. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*. 15. 1929-1958.